A Fast Track Architecture for UDP/IP and TCP/IP

Roy Chua
rchua@eecs.berkeley.edu

MacDonald Jackson
trey@cs.berkeley.edu

Marylou Orayani
marylou@cs.berkeley.edu

May 9, 1995

Abstract

Poor TCP/IP and UDP/IP implementations have been attributed as a factor to the observed lack of performance of remote applications given the speeds of modern communication networks. We discuss some of the factors that cited as contributors to the protocols' poor performance. We then discuss the fast track architecture for UDP/IP and TCP/IP which improves protocol performance by optimizing for the common rather than the general case and by bypassing the operating system completely thereby reducing any overhead that may be incurred while in the operating system. We then present our implementation of a fast track UDP/IP along with proposals for fast track TCP/IP implementations.

1 Introduction

Many researchers have observed that the performance of remote applications have not kept pace with modern communication network speeds. Part of this imbalance is attributed to the protocols used by the remote applications, namely, UDP/IP and TCP/IP. It is now widely believed that the problems with these protocols are not inherent in the protocols themselves but in their particular implementations [Dalt93, Whet95]. The following factors are cited as contributors to the poor performance of UDP/IP and TCP/IP:

Multiple copying of data This is considered the biggest bottleneck. Receiving a packet requires two copies: one copy from the network interface to the kernel and another one from the kernel to the receiving application. Sending a packet also requires two copies: one from the sending application to the kernel and another one from the kernel to the network interface. Measurements in [Kay93] show that moving packet data takes up 9% of TCP processing time and 16% of UDP processing time.

Protocol layering Layering facilitates protocol design but not implementation. It causes bad buffering decisions which generate unnecessary memory traffic [Jacob93]. Packets are forced to go from application buffers to socket buffers to kernel buffers and finally to the network interface buffers. Layering also hinders parallelism [Jacob93] forcing applications to call generic routines instead of protocol-specific ones causing a bottleneck at the generic routines.

Optimization for the general rather than the common case Because of the generality of the code, a lot of time is spent processing packet headers, calculating checksums in the kernel and managing buffers [Whet95]. This was acceptable due to the unreliability of the internetworks within which the protocols were operating but with modern networks, most packets arrive in sequence and are error-free [Jacob93].

Checksum computation It is known that checksum calculation is expensive. In addition, it requires an additional perusal of a packet's header and data. Measurements in [Kay93] show that checksumming of TCP/IP packets take up only 7% of TCP processing time while it takes up 25% of UDP processing time. This is attributed to the larger sizes of UDP packets. Analysis of a FDDI network traffic in a university LAN [Kay93] calculated a median TCP packet size of 32 bytes and a median UDP packet size of 128 bytes.

Kernel buffer manipulation The kernel buffers used for network packet processing are the memory buffers (mbufs) and the socket buffers. According to [Kay93], mbuf and socket buffer processing take up 18% of TCP processing time and 15% of UDP processing time.

Context switches A context switch involves a process blocking itself (e.g., via the `sleep` system call), waking up another process and restarting it. According to Kay et. al. [Kay93], context switches take up about 5% of TCP processing time and 4% of UDP processing time [Kay93].

Error checking Error checking time is largely taken up by the validation of arguments to system calls. This takes up 5% of TCP processing time and 7% of UDP processing time.

Interrupts When a packet arrives, two interrupts are generated: a hardware interrupt generated by the network interface to indicate that a packet has arrived and a software interrupt generated by the network interface device driver to tell IP that a new packet has arrived. Generating a software interrupt and handling the interrupt by dequeueing the incoming packet and calling IP to handle it takes up 2% of processing time for both TCP and UDP.

In [Whet95], a fast track architecture for UDP/IP and TCP/IP was proposed. Its main aim is to improve protocol performance by optimizing for the common case and by bypassing the operating system (OS) completely, reducing any overhead that may be incurred while in the OS. Given network interface support, the fast track can also limit the copying of data to one. The architecture distinguishes between two types of packets: common case and non-common case (common case criteria are discussed in detail in Section 3) and only processes the common case packets. A packet buffer is shared between the application and the network interface device driver. When a common case packet arrives at the receiving host, the device driver copies the packet into the packet buffer it shares with the application. This effectively demultiplexes the packet directly to the application and bypasses the operating system completely. To send a common case packet, the sending application directly copies the packet into the buffer it shares with the device driver and calls a device driver routine to send the packet to its destination.

The fast track architecture is applicable to a wide range of network interfaces [Whet95] and can coexist with other UDP/IP and TCP/IP implementations.

Section 2 discusses problems reported and solutions proposed by researchers in the area. Section 3 presents the goals of the fast track architecture. Section 4 describes the fast track implementation of UDP/IP along with the current UDP/IP implementation. Section 5 is a discussion of fast track and the current TCP/IP implementation. Section 6 is an analysis of the fast track UDP/IP implementation. Section 7 presents difficulties we encountered during implementation. Section 8 presents our conclusions and summary.

2 Previous Work

Various researchers have proposed and implemented solutions to overcome the problems with UDP/IP and TCP/IP implementations discussed in Section 1. In [Part94], Craig Partridge showed ways to speed up UDP/IP implementation one of which is to calculate the checksum while copying the data. Van Jacobson [Jacob93] has implemented a high performance TCP/IP by reducing the number of interrupts generated for each packet sent or received, by touching any piece of data exactly once, and by doing away with layering and maximizing parallelism. His code only requires 37 instructions and 7 memory references to forward an IP packet, and less than 60 instructions and 22 memory references to process a TCP packet.

Dalton et. al. [Dalt93] modified TCP/IP to support the Afterburner card which they designed and built. The card provides buffers that can be directly accessed by the operating system so that data is copied only once: from the on-card buffers to the application buffers for receiving and vice-versa for sending. They achieved a throughput of 210 Mbps with 14 KB packets.

Brustoloni [Brust94] used exposed buffering to avoid copying of data altogether. A special area in kernel memory is allocated specifically for these exposed buffers so that both the applications and the network interface can share access to them. When a packet is received, it is placed directly in these buffers and the application reads the packet directly from them. To send a packet, the application writes into the buffers and the network interface sends it from the same buffers. This was implemented in the Mach operating system on a host connected to an ATM LAN. They achieved a throughput of 48.4 Mbps with 64 KB datagrams and 48.5 Mbps with 64 KB sequenced packets.

Kay et. al. [Kay93] suggested avoiding checksumming for both TCP and UDP when the communicating hosts are on the same LAN provided the LAN supports a hardware CRC. They also stated that reducing TCP protocol-specific processing time is useful but will require a major modification of TCP implementation. It is unclear whether the effort required is worth the performance improvement that will be gained.

Whetten et. al. [Whet95] proposed an implementation for a fast track receiver and sender for both TCP/IP and UDP/IP. We initially tried to implement the fast track TCP/IP receiver but encountered difficulties with the proposed implementation. The proposal did not take into account that for both the fast track and non-fast track TCP/IP implementations to coexist, they must be able to share state information about TCP/IP connections. We had a few options:

- Remove the requirement that fast track and non-fast track TCP/IP must coexist.

  This is infeasible since fast track setup requires assistance from non-fast track TCP/IP. In addition, non-common case TCP/IP packets will never be unprocessed.

- Modify TCP/IP to be able to more readily share state information with fast track TCP/IP.

  Given the complexity of TCP/IP code, we believed this would require too much work given the amount of time we had.

- Implement fast track UDP/IP instead.

  This is the option we eventually chose because of the simplicity of UDP/IP relative to TCP/IP.

In the remaining sections of the paper, we first present the fast track goals for both UDP/IP and TCP/IP and follow this with a discussion of fast track UDP/IP architecture and implementation. This is then followed by a discussion of the fast track TCP/IP architecture and a number of proposals for its implementation. We then present an analysis of our fast track UDP/IP implementation.

3 Fast Track Goals

For both UDP/IP and TCP/IP, the goals of the fast track architecture are to optimize for the common case (vs. the general case), to minimize operating system overhead and to minimize data copying. A UDP/IP packet is considered common case for the receiver if it satisfies the following criteria:

- It uses the correct IP version (4).
- It has the correct header length (5).
- It is not fragmented.
- It is destined for this host.
- It is error-free.
- A socket already exists for the packet.

The criteria for a TCP/IP packet are identical with the following additions:

- It is not a control packet.
- It has the correct sequence number (i.e., the packet arrived in order).
- It has no variable length options.

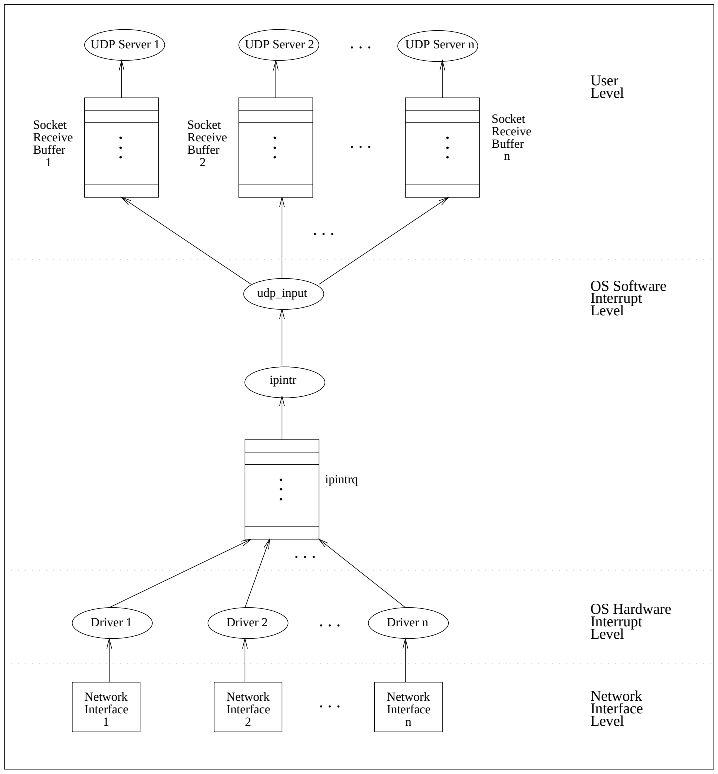

Figure 1: Current UDP/IP Receiver

The criteria for a common case UDP/IP packet for the sender are:

- No packets waiting to be sent.
- No fragmentation required.
- Already have an outgoing socket.
- Have already resolved the outgoing link address using ARP

The criteria for sending a TCP/IP packet are identical with the following additions:

- Window size must be big enough and have space.
- Must have a valid connection

Though treated separately in this paper, fast track UDP/IP and TCP/IP can and should be combined in one implementation. The separation was done for simplicity and clarity.

4 Fast Track UDP/IP

In this section, we will first describe the current implementation of the UDP/IP receiver and then contrast it with the fast track implementation. This is followed by a similar discussion of the UDP/IP sender.

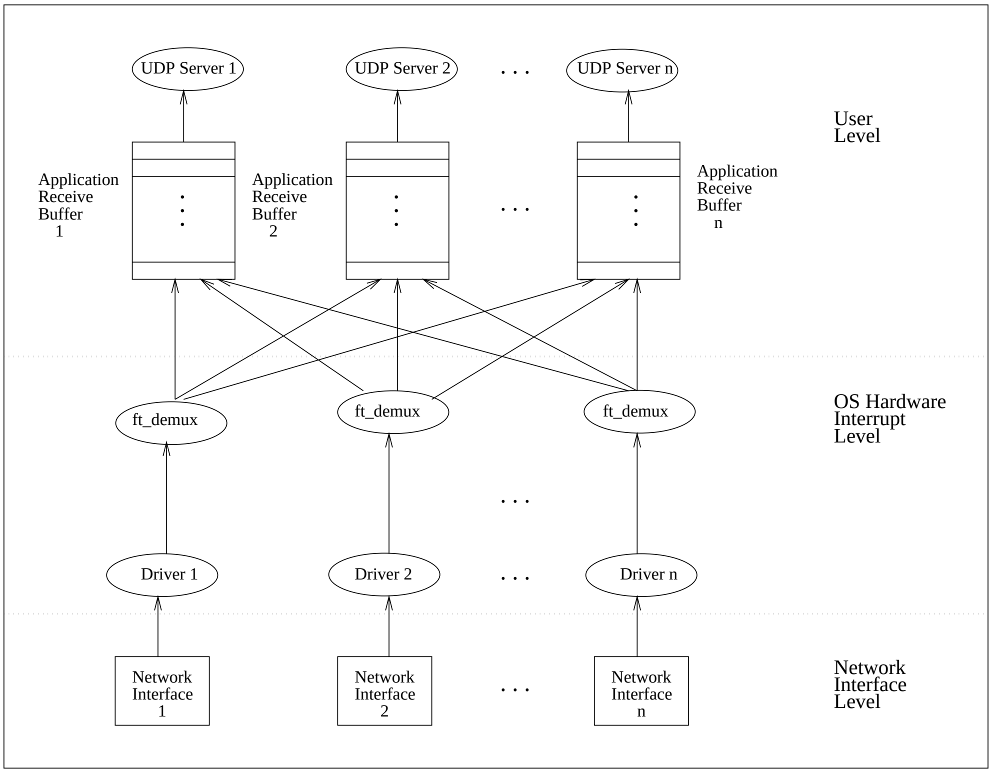

Figure 2: Fast Track UDP/IP Receiver

4.1 Current UDP/IP Receiver

Figure 1 shows a diagram of the current UDP/IP receiver. When a packet arrives at the network interface, it generates a hardware interrupt which activates the interface's device driver. If the packet is an IP packet, the driver puts it into the IP input queue, `ipintrq`, and generates a software interrupt to inform the IP interrupt handler, `ipintr`, that a new IP packet has arrived. Based on the packet's protocol type (specified in the IP header), IP demultiplexes the packet to the appropriate protocol. If it is a UDP packet, `ipintr` invokes the UDP input function, `udp_input`, to process the incoming packet. `udp_input` first validates the incoming packet by verifying its length and by calculating its checksum. `udp_input` then determines the packet's recipient application and puts the packet into that application's socket receive buffer. Two interrupts are generated, the hardware interrupt generated by the network interface and the software interrupt generated by the device driver. One can clearly see the bottleneck at `ipintr` and `udp_input` due to layering.

4.2 Fast Track UDP/IP Receiver

We implemented the receiver on a HP-UX 9000/700 workstation running the HP-UX OS version A.09.01.

Figure 2 shows a diagram of the fast track UDP/IP receiver. When a packet arrives at the network interface, it generates a hardware interrupt which activates the interface's device driver. The driver then calls the fast track demultiplexer, `ft_demux`, which determines if the packet is common case or not. If it satisfies all the common case criteria enumerated in Section 3 above, `ft_demux` copies the packet from the network interface to the appropriate application receive buffer (which replaces the socket receive buffer). Otherwise, the packet is placed in the appropriate protocol input queue and the appropriate software interrupt is generated. Only one interrupt is generated, the hardware interrupt generated by the network interface. Except for the brief stay in the demultiplexer, the operating system is completely bypassed. In addition, no bottlenecks exist.

Destination Port	Virtual Space Id	Virtual Address Space Pointer	Application Receive Buffer Pointer
(2 bytes)	(4 bytes)	(4 bytes)	(4 bytes)

Figure 3: Demultiplexer Table Entry Format

```
create socket
while more packets to receive
   call ft_recvfrom
```

Figure 4: Fast Track Application Server Algorithm

4.2.1 Fast Track Demultiplexer Table

The key component of the fast track receiver is the fast track demultiplexer, `ft_demux`. The key data structure used by `ft_demux` is the demultiplexer table, `DemuxTable`. Each socket that uses the fast track has an entry in `DemuxTable`. Each entry contains the port used by the receiving application (destination port), the application's virtual space id, a pointer to its virtual address space, and a pointer to its receive buffer. The destination port essentially identifies the receiving application because each server socket can only be used by one server. The virtual space id and pointer are needed by the demultiplexer to correctly access the application's receive buffer. The receive buffer replaces the socket receive buffer to minimize (if not eliminate) the use of kernel buffers. Figure 3 shows the format of an entry and the number of bytes each field uses. `ft_demux` also uses hint pointers to the most recently used entries in the demultiplexer table to take advantage of the fact that most packets demultiplex to the same place [Jacob93].

4.2.2 Fast Track Recvfrom

To enable fast track UDP/IP receiving, the application only has to call `ft_recvfrom` instead of `recvfrom`. Calling `ft_recvfrom` is just like calling tt recvfrom except that an additional parameter is given, the application's address as specified by bind. Two buffers are needed by an application to receive packets: a receive buffer used for queueing packets destined for the application and a packet buffer used to store a packet from the receive buffer. The receive buffer is a first-in- first-out (FIFO) buffer. The packet at the beginning of the buffer is moved from the receive buffer into the packet buffer. Applications can only directly read packets from the packet buffer. To minimize any additional work the application has to do, space needed for the receive buffer is allocated by `ft_recvfrom`. The packet buffer is supplied by the application as a parameter to `ft_recvfrom` (just like in `recvfrom`). When called for the first time, `ft_recvfrom` creates and inserts an entry for the specified socket (the destination port) into the demultiplexer table. `ft_recvfrom` also calls `recvfrom` to receive the socket's very first socket. Subsequent calls to `ft_recvfrom` will first poll the application's receive buffer until a packet is placed in it by `ft_demux`. For simplicity, polling is used instead of select to detect data in the receive buffer. Each packet placed by `ft_demux` in the receive buffer can be one of two types: common case (fast track) or non-common case (slow track). If the packet at the beginning of the buffer is slow track, this means that actual packet data is not in the buffer since `ft_demux` does not process slow track packets. To process a slow track packet, `ft_recvfrom` calls `recvfrom` to receive the packet using the conventional operating system channels. If the packet is fast track, it is copied from the application's receive buffer into its packet buffer. Figures 4 and 5 show the algorithms for the fast track application server and `ft_recvfrom`, respectively.

```
if called for the first time
    call recvfrom
    allocate space for receive buffer
    add an entry to demultiplexer table
else
    poll receive buffer for data
    if slow track packet at head of buffer
        call recvfrom
    else
        copy packet into specified packet buffer
return number of bytes received
```

Figure 5: `ft_recvfrom` Algorithm

4.2.3 Fast Track Demultiplexer

When an incoming packet arrives, `ft_demux` is activated by the hardware interrupt generated by the network interface to indicate the packet's arrival. `ft_demux` first checks that the packet is a UDP/IP packet. If it is not, it is simply sent back to the OS. Otherwise, `ft_demux` checks that the packet contains the correct IP version number (4) and the correct header length (5). If it does not, it is sent back to the OS. Otherwise, `ft_demux` consults the demultiplexer table, `DemuxTable`, for an entry that has the same source/sender IP address as the incoming packet, the same source and destination ports as the incoming packet, and the same protocol (UDP) as the incoming packet. If a matching entry is found, the packet is copied into the application receive buffer specified in the `DemuxTable` entry. To speed up searches, hint pointers are used to keep track of the most recently used table entries. These table entries are first consulted and if no match is found using the hint pointers, the entire table is searched for matching entries. Figure 6 shows the `ft_demux` algorithm.

4.3 Current UDP/IP Send

Figure 7 shows a diagram of the current UDP/IP sender. When a process, called a UDP Client, sends a packet using the system call sendto, the destination socket address structure and user data are copied into an mbuf chain. This mbuf chain is then passed to the protocol layer, first `udp_output`, then `ip_output`, which prepend the appropriate headers. `udp_output` calls `if_output` to pass the datagram to the network interface layer, where the packet is placed in a queue to be transmitted to the network. Note that user data does not wait in any queues until it reaches the device driver.

4.4 Fast Track UDP/IP Sender

We implemented the sender on the same machine as the receiver, an HP-UX 9000/700. Figure 8 shows the control flow of the fast track UDP/IP sender. The UDP Client sends a datagram by calling `ft_sendto`. If the packet meets the common case conditions as described in section 3 above, the data is copied to a mbuf chain with the UDP and IP headers prepended, and the packet is sent directly to the network interface to be sent out to the network. If the packet fails any of the conditions control is passed to sendto.

The fast track UDP code reduces some overhead by caching header fields and cutting out two function calls. `ft_sendto` has a `MuxTable`, which performs the same role as the `DemuxTable` does on the receive side. The table is indexed by the UDP Client's source and destination sockaddr's. Port pairs alone will not resolve the ambiguity because the UDP Client may send to more than one destination through the same socket, and each of those destinations may use the same port. For the same reason, socket file descriptors will not work. The locations of the sockaddr's will be unique.

```
if packet is not IP
    return to OS
if packet is not UDP
    return to OS
if packet's IP version is incorrect
    return to OS
if packet's header length is incorrect
    return to OS

for each hint pointer
    if tableEntry[hintPointer].destPort == packet.destPort
       copy packet into tableEntry[hintPointer].recvBuf
endfor

if no matches found
   for each table entry
        if tableEntry[entryNo].destPort == packet.destPort
          copy packet into tableEntry[hintPointer].recvBuf
    endfor

if no matches found
   return to OS
```

Figure 6: `ft_demux` Algorithm

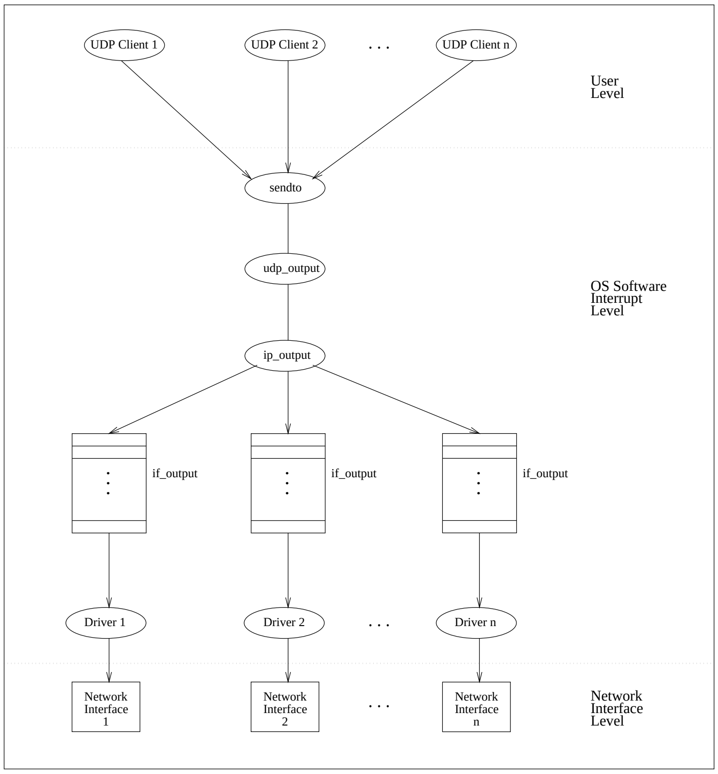

Figure 7: Current UDP/IP Sender

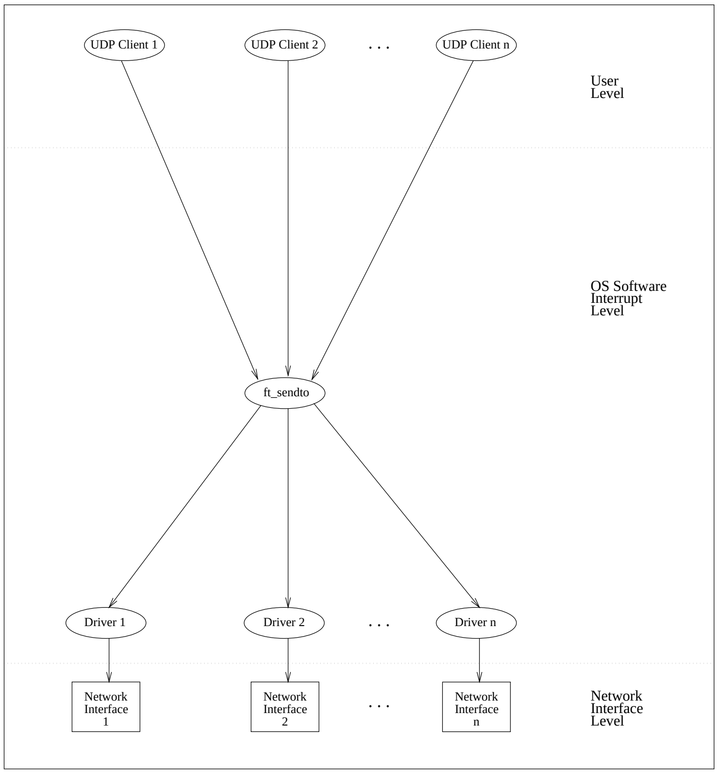

Figure 8: Fast Track UDP/IP Sender

Cached in the `MuxTable` are both the UDP and the IP headers. When `ft_sendto` receives a packet from a new UDP Client, the `MuxTable` entries are filled in, and the packet is sent to the network using sendto (because this is a special case of not meeting the fast track criteria). When a packet is received by `ft_sendto` from a known UDP Client, the headers are copied into the mbuf using the cached values. The only headers that need to be updated are the IP header checksum, the IP length, the UDP length, and the UDP checksum. Currently the UDP checksum is not computed. See figure 9 for the algorithm.

Another optimization is the assumption that there are no packets waiting to be sent in the network driver. This facilitates speedy data transmission. Currently the network interface does not support such a check.

5 Fast Track TCP/IP

In this section, we first give a brief description of the current implementation of the TCP/IP receiver and follow it with a general description of the fast track TCP/IP receiver. We then present a number of proposals on how a fast track TCP/IP receiver can be implemented. This is followed by a similar discussion of the sender.

5.1 Current TCP/IP Receiver

Figure 10 is a diagram of the current implementation of the TCP/IP receiver. When a packet arrives at a network interface, that interface generates a hardware interrupt which activates its driver to process the incoming packet. If the packet is an IP packet, the driver puts it into the IP input queue, `ipintrq`, and generates a software interrupt to activate the IP interrupt handler, `ipintr`. If the packet is a TCP packet, `ipintr` then calls the TCP input function, `tcp_input`, to further process the packet. `tcp_input`

```
if packet is not IP
    sendto(packet)
if packet needs to be fragmented
    sendto(packet)

for each hint pointer
   if ((MuxTable[hintPointer].sorc == uap->from) &&
       (MuxTable[hintPointer].dest != uap->to))
     header = MuxTable[sendHints[i]].header;
endfor

if no matches found
   for each table entry
      if ((MuxTable[EntryNo].sorc == uap->from) &&
          (MuxTable[EntryNo].dest != uap->to))
        header = MuxTable[sendHints[i]].header;
   endfor

if no matches found
   add new entry

if ARP resolved
    send to network interface
else
   sendto(packet)
```

Figure 9: `ft_mux` algorithm

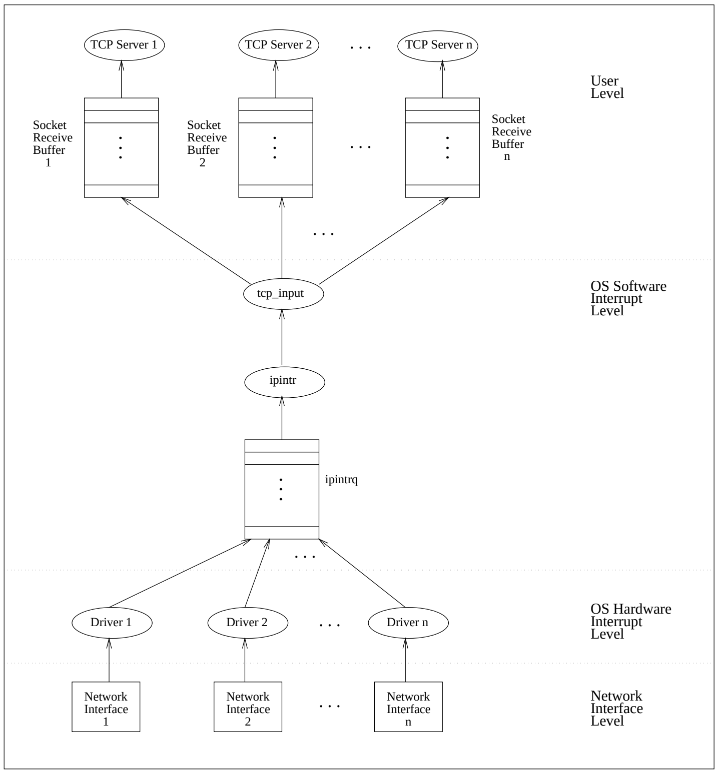

Figure 10: Current TCP/IP Receiver

first validates the incoming packet by calculating its checksum and then compares it with the checksum included in the TCP header. It then calculates the amount of space available in the `socket` receive buffer (this available space is called the receive window) for the receiving application. The received packet is then trimmed (if necessary) to fit into the receive window. `tcp_input` then queues the trimmed packet into the application's socket receive buffer. The application obtains the packet from this receive buffer via one of the socket receive functions, e.g., recv and read. In addition to queueing the buffer into the appropriate socket receive buffer, `tcp_input` performs some critical functions to correctly maintain the connection's state. These include updating the receive and congestion windows if needed, determining the sequence number of the next expected packet, remembering the packets received but not yet acknowledged, and sending acknowledgements for received packets. This state information maintenance is what makes it much more complicated than UDP.

In the process of receiving an incoming TCP/IP packet, two interrupts are generated: the hardware interrupt generated by the network interface and the software interrupt generated by `ipintr`. As in UDP, a bottleneck exists at `ipintr` and `tcp_input` due to layering.

5.2 Fast Track TCP/IP Receiver

Implementation of the fast track TCP/IP receiver is structurally identical to its UDP/IP counterpart. Figure 10 shows a diagram of the implementation structure.

When a packet arrives at a network interface, it generates a hardware interrupt to activate its device driver so it can process the incoming packet. The driver then calls the fast track demultiplexer, `ft_demux`, to demultiplex the packets to the appropriate application receive buffer. As in the fast track UDP/IP receiver, the demultiplexer is the key component of the receiver (see Section 4.3 for details).

To receive fast track TCP/IP packets, the application must invoke the usual sequence of calls: `socket`, `bind`, `listen`, and `accept`. `accept` creates a new socket for each new connection accepted. It is at

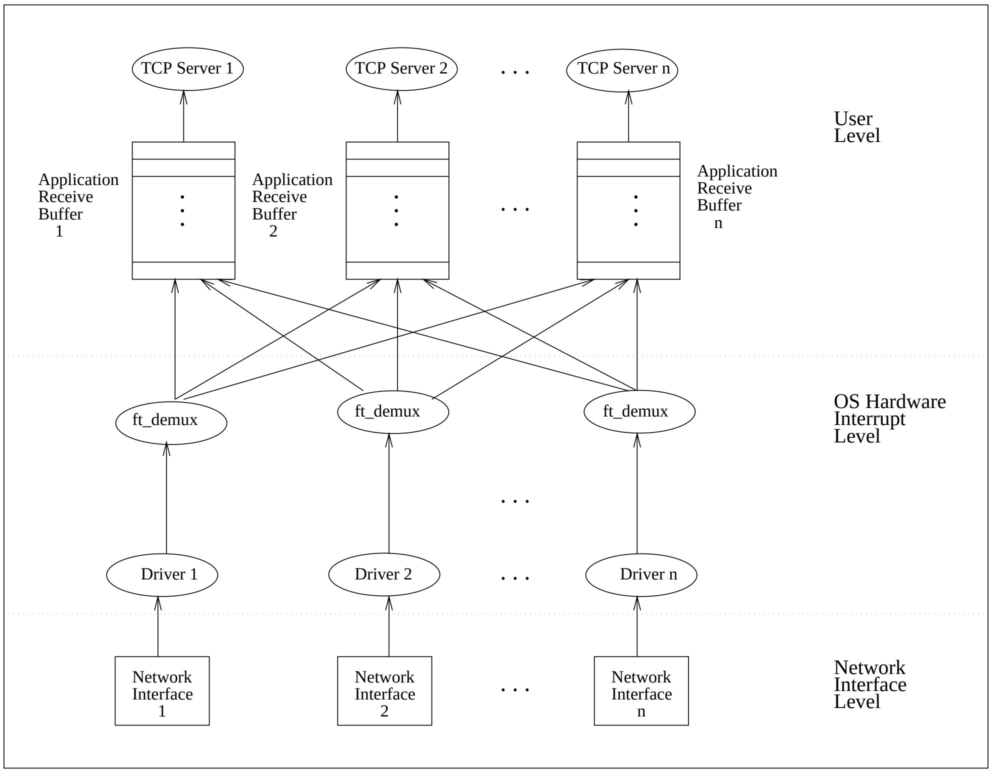

Figure 11: Fast Track TCP/IP Receiver

this point where a demultiplexer table entry is created for the new socket and added to the demultiplexer table. Instead of calling read or recv, the application must call `ft_recv` which receives fast track TCP/IP packets processed by `ft_demux`. `ft_recv` behaves just like `ft_recvfrom` (used for UDP/IP packets) except that it has to perform additional functions to maintain state information for the connection and share this information with the non-fast track TCP/IP for both to function correctly. This makes `ft_recv` much more complicated than `ft_recvfrom`.

5.3 Current TCP/IP Sender

The control and data flow of the current implementation of the TCP/IP sender is shown figure 12. When a TCP Client calls send to send a packet, the data flows much as it does in the UDP case. The data is copied to an mbuf chain. This mbuf chain is passed down to the network interface level through `tcp_output` and `ip_output`, both of which prepend the proper headers. `ip_output` passes the mbuf to the network interface by calling `if_output`.

Unlike UDP, sending data using TCP can result in queueing user data before the network drivers. Because TCP is a connection-oriented service and is based on a sliding window scheme, a large amount of state must be managed in order to guarantee the service it provides. `tcp_usrreq` is the function which handles this "bookkeeping", including connection management, congestion window size, timers and associated timing computation, and data retransmission. For a detailed description of `tcp_usrreq` see [Wrig95]. Due to this high overhead and to lost packets, user data can be queued at both the TCP and the network interface layers.

5.4 Fast Track TCP/IP Sender

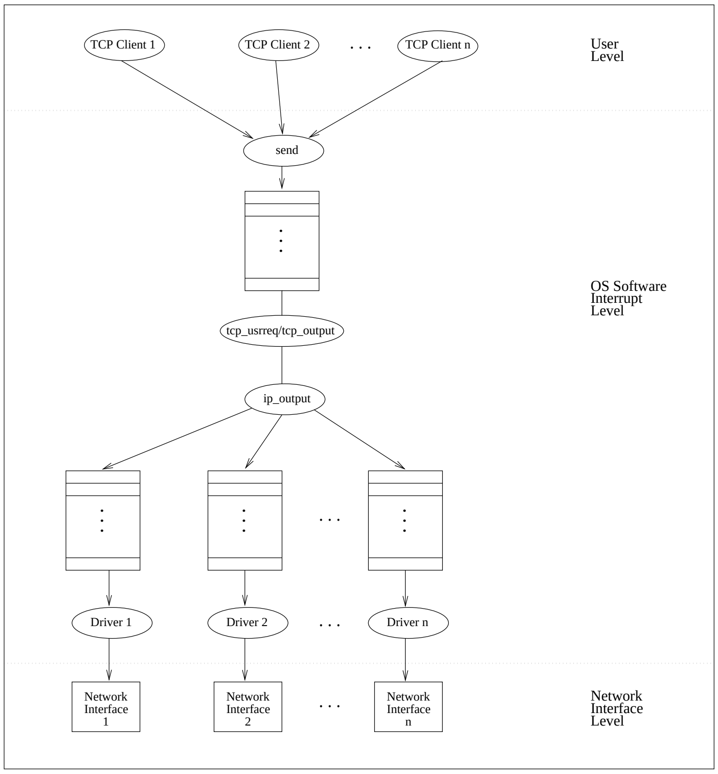

Figure 12: Current TCP/IP Sender

Figure 13 is markedly different from its counterpart describing the current TCP/IP sender. Upon receiving a packet to send, `ft_send` would check the packet to determine if it is common case, and if so retrieve the appropriate headers from the cache. The only headers necessary to compute are the IP length and header checksum, and the TCP sequence number, acknowledgment number, window information, and checksum. The only additional overhead is incurred by calculating and setting the retransmission timer.

In order for a TCP Client to send data it must invoke the usual system calls: `socket`, `bind`, and `connect`. During the `connect` call a new entry in the `MuxTable` would be created for this connection. To send data using the fast track the client would use the system call `ft_send`. Because `ft_send` coexists with the original implementation of TCP, they must share state in order to perform reliably. This need for sharing state between the fast track and the original will result in a very hairy implementation.

5.5 Fast Track TCP/IP Receiver Implementation Proposals

Because of its guarantee of reliability, TCP has to maintain some state while a connection exists between two sockets. On the receiving end, this includes the sequence number of the next incoming packet, the number of packets received so far and still unacknowledged, and the receive and congestion window sizes (for congestion control). In addition to maintaining state, the fast track receiver also has to inform the non-fast track, or slow track, TCP/IP receiver of the current connection state. This is where most of the complexity lies. If all incoming TCP/IP packets were fast track, implementation would be much simpler since all the fast track receiver has to do is maintain its own data structures. However, in order to correctly process slow track packets, the slow track TCP/IP receiver must be informed of what has transpired during fast track processing. The bottom line is that both fast track and slow track TCP/IP receivers must be able to share state information for them to function correctly since their actions are based on the connection's state. Several ways of implementing this are proposed in the following sections.

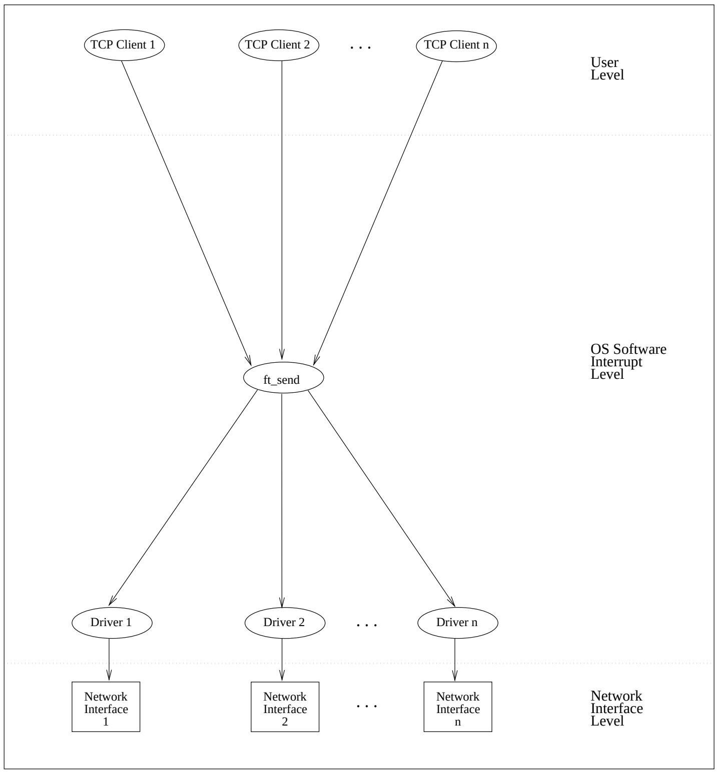

Figure 13: Fast Track TCP/IP Sender

5.5.1 State Information Sharing

One way that both fast track and slow track TCP/IP can share information is by sharing data structures. This would require a reimplementation of TCP/IP so that it can more readily share data structures with its fast track counterpart. This can be done by having a global data structure that contains the minimal information needed by the slow track TCP/IP to function correctly. This information includes the sequence number of the next incoming packet, packets received but not yet acknowledged and the sizes of the receive and congestion windows. Each time fast track TCP/IP processes a packet, it updates this shared data structure. When slow track TCP/IP is reactivated to process a slow track packet, it first consults the shared data structure to update its state information and proceeds to process the incoming packet which includes updating the shared data structure. This facilitates switching between fast and slow tracks. To obtain further performance gains from fast track, we can defer the updating of items not used by fast track until a slow track packet comes in. Before switching to slow track, we would then update all the necessary fields to guarantee correctness. In addition to sharing data structures, we have to watch out for retransmission timers and RTT estimates. The issue of sharing retransmission timers between fast and slow track is an unresolved and will require careful design and thought.

5.5.2 User-Level TCP on Top of Fast Track UDP/IP

A user-level TCP can be implemented on top of fast track UDP/IP. This TCP can either replace the slow track TCP completely or coexist with it. If the two coexist, then state information must again be shared between the two. The shared information should reside in user space to prevent the user-level TCP from crossing protection boundaries (from user to kernel level) just to access this information. If the user-level TCP replaces the slow track TCP, then all TCP processing will be done at user-level. Though this may lead to a simpler implementation, performance costs may be incurred since TCP processing can now be preempted by any other process.

5.5.3 Rewrite Current TCP Implementation to Implement Fast Track Ideas

This would be a rather drastic approach to take, but we believe that the end-result of this will be more efficient than the above two. It should be possible to take the current TCP implementation and restructure it to implement fast track ideas. For example, during the network hardware interrupt, some processing could be done to determine if this is a fast track packet etc. Some data structures can already be updated at this point, since we already know what kind of packet it is, the size, whether or not it is fragemented etc. If it turns out that the packet is not fast track, we can enqueue it and schedule a software interrupt later. Since we've already done some of the work at this hardware interrupt level, we have less work to do later. If the packet turns out to conform to our fast track specifications, we can deliver the packet to user space immediately.

6 Analysis of Fast Track UDP/IP Implementation

In section 4.2, we discussed how we implemented the Fast Track UDP/IP (FT UDP/IP) code within the HP-UX kernel. The code has been tested with both multiple senders and multiple receivers. We have sucessfully tested the code with thousands of packets of varying sizes with multiple senders and one receiver, as well as with multiple receivers and multiple senders. We have analyzed the code in assembly and have instrumented our code within the kernel to calculate the time required for fast track implementation. As for the existing UDP/IP implementation, [Part94] describes times for i386 and Sparc 1 machines, but it would not be possible to try to rescale those figures for a HP 9000/700 series machine.

6.1 Receiver

The main costs involved when processing an incoming packet are in the `ft_demux` call. Counting assembly instructions and assuming that the packet satisfies fast track conditions and fits into one mbuf, we get 317 assembly instructions. The 317 includes 1 call to bcopy, 2 to `ft_copyin`, 3 to `ft_copyout`, 1 call to `m_free`, 1 call to `splimp` and 1 call to `splx`. `ft_copyin` is 51 instructions in total, but since we know that the location we are copying from is locked in memory, only 47 of those instructions are executed. The 47 instructions include 1 call to `ulbcopy`. `ft_copyout` is 31 instructions in total (including two function calls, one to `lacc`, and one to `ulbcopy`), and number of instructions in `lacc` is about 100.

`ulbcopy` can be broken down into the following:

Ulbcopy Setup	36 instructions
Quad Words	
Copying Quad	9 instructions/quad word
Check Words Left	5 instructions
Remaining Words	
Init	1 instruction
Copying Word	3 instructions/word
Check Last Word	5 instructions

We use `ft_copyout` to send the packet to user space. So, if we assume a small packet, we have $36 + 9 + 5 + 1 + 3 + 5$ which is 59 instructions. Also, `ft_copyout` is used to copy out the queue entry and the tail, and we have 60 instructions for each copy. So, the total copy in cost is $2 * (47 + 60) = 214$. And the copy out cost is $3 * (31 + 100 + 60) = 573$.

`bcopy` is almost the same as `ulbcopy`, and we use it to zero out one long word, so we have here another 50 instructions.

splimp is a call to splx, so we have $2 *$ splx calls. A count of the assembly code gives a figure on the order of 50 instructions.

m_free has 103 instructions, including 2 calls to splimp/splx, 1 call to hdl_vpfn, 1 call to spinlock_usav 1 call to spinunlock_usav and 1 call to kfree. An examination of the code indicates that m_free, in addition to its 103 instructions, will incur about $2 * 50 + 50 + 50 + 100 = 300$ extra instructions as a result of the function calls.

So, the approximate total number of instructions involved in processing a small fast-track packet is $317(\texttt{base}) + 214(\texttt{copyin}) + 573(\texttt{copyout}) + 50(\texttt{bcopy}) + 100(\texttt{spl}) + 403(\texttt{m\_free}) = 1657$.

The machine we are using is a RISC machine, and some of the assembly instructions are expanded into 2 to 3 actual machine instructions. Assuming this happens for 10% of the instructions, we get around 1823 instructions. Rounding this up to around 2000 instructions, we see that on a fast machine (e.g. a RISC machine with a $20ns$ cycle time, issuing approximately 1 instruction per second) code takes $40\mu s$ to run.

Of course, calculating assembly instructions is not as accurate as instrumenting the actual code. As such, we instrumented the ft_demux code within the kernel and added kernel gettimeofday calls. In order to factor out the gettimeofday overhead, we ran separate test cases to work out the time overhead. It turns out that a call to the kernel version of gettimeofday takes $4.8\mu s$ (approx $5\mu s$). Trying out various packet sizes and taking averages, the following is a sample of the time ranges we see (excluding the gettimeofday overhead):

Data Size (bytes)	Time Taken (μs)
2	40
15	48
30	51
55	50
108	52
273	53

As we can see, the copy out overhead is not really substantial. This can be explained by the HP's ability to copy huge amounts (quad words) of data quickly.

Instrumenting the current UDP/IP implementation has proven intractable due to the complexity of the kernel code. So far, we have been unable to find numbers in the literature for this architecture and operating system. [Dalt93] discusses some figures for TCP/IP on a similar architecture, but the figures were obtained based on analysis rather than actual timing results.

6.2 Sender

Given that the sender code is still not working correctly, we have not analyzed in detail the operation of the sender. In addition, due to the relative simplicity of the original UDP/IP sender, not much gain is expected from our fast track sender. As a result, we concentrated our efforts on getting the fast track receiver working, knowing that this is where we would obtain the greatest improvements.

7 Implementation Difficulties

We have encountered numerous difficulties in implementating fast track UDP/IP. Though the algorithms we used were straightforward, unexpected details came up to complicate the implementation process. These are discussed in detail below.

Kernel code debugging Debugging time was excruciatingly slow due to the lack of a kernel debugger. Everytime the code is modified, the kernel has to be rebuilt and the system rebooted. On a HP-UX 9000/700 workstation, this took about 5 minutes. In addition, when the code is incorrect, the system crashes (not just a program). Recovery from crashes took about 10 minutes.

Lack of HP-UX expertise We had no access to anyone or anything that would have helped us in understanding some peculiar operating system behavior we encountered such as mysterious system crashes and HP-UX idiosyncracies (e.g., how was the kernel stack implemented?). Our main source of information was the source code which, as expected, was not easy to understand.

Complexity and volume of kernel and network code This was mostly a problem at the beginning causing a fairly steep learning curve.

Lack of experience in kernel hacking In addition to raising the learning curve, this caused us to underestimate the amount of time needed to accomplish our goals.

Complex interaction between kernel and user level processes System crashes occur for inexplicable reasons from time to time. The error messages used to indicate a *data page fault on interrupt stack*. This happened due to our not accessing the right virtual address space (vas) during copying between kernel and user spaces. After fixing this problem, we still had system crashes, though not as often. This time around, the error messages indicated a *TLB miss on interrupt stack*. We have, so far, been unable to resolve this problem. However, we believe that a thorough understanding of HP's kernel implementation and system architecture will allow us to solve this problem.

8 Summary and Conclusions

Various factors have been attributed to the poor performance of the current implementations of the UDP/IP and TCP/IP protocols. These include the multiple copying of data done by these protocols which dominates the processing time, the protocol layering which facilitates design but not implementation, optimization of code for the general rather than the common, checksum computation which may not be necessary in modern networks, kernel buffer manipulation, context switches, error checking, and interrupts. A fast track architecture has been proposed to improve protocol performance by optimizing for the common rather than the general case and by bypassing the operatins system completely, reducing any overhead that may be incurred while in the OS. We have initially attempted to implement a fast track TCP/IP but quickly encountered difficulties due to an important issue that the fast track architecture proposal failed to take into account, namely, the necessity for both fast and slow track TCP/IP implementations to share connection state information. This introduced complexities that required much more work that could be done in the amount of time we had. We then went on to implement a fast track UDP/IP receiver on a HP 9000/700 workstation running the HP-UX OS. Though seemingly straightforward, we encountered numerous difficulties during implementation, a big factor of which was due to the excruciatingly slow debugging cycle involved in kernel code implementation. Despite this, we were able to successfully implement a fast track UDP/IP receiver which proves the viability of the fast track architecture for UDP/IP. However, in the process of testing, we ran into system crashes of which the causes are still unkown at publication time. Further work is needed to determine the cause(s) for these crashes to make the receiver more robust. In addition, it was difficult to instrument HP-UX's UDP/IP implementation to obtain timing information due to to the complexity of the code which spans multiple OS layers. Though seemingly straightforward, modification of an existing protocol requires quite a bit of effort. We attribute this difficulty to the poor design and implementation and the lack of flexibility of existing protocols and operating system implementations.

References

[Brust94] J. C. Brustoloni. "Exposed Buffering and Sub-Datagram Flow Control for ATM LANs". *Proceedings of IEEE Local Computer Networks*, pp. 176-185, October, 1994.

[Dalt93] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. "Afterburner". *IEEE Network Magazine*, pp. 36-43, July, 1993.

[Intel92] *32-Bit Local Area Network Component User's Manual*. Intel Corporation, 1992.

[Jacob93] V. Jacobson. "A High Performance TCP/IP Implementation". *Gigabit-per-Second TCP Workshp, Center for National Research Initiatives*, March, 1993.

[Kay93] J. Kay and J. Pasquale. "The Importance of Non-Data Touching Processing Overheads in TCP/IP". *Proceedings of SIGCOMM '93*, September, 1993.

[Lef90] S. Leffler, M. McKusick, M. Karels and J. Quarterman. "The Design and Implementation of the 4.3BSD UNIX Operating System". Addison-Wesley Publishing Co., 1990.

[Mart94] R. Martin. "HPAM: An Active Message layer for a Network of HP Workstations". Work in progress.

[Mart95] R. Martin. Private communication.

[Part94] C. Partridge and S. Pink. "A Faster UDP". Work in progress.

[Stev90] W. R. Stevens, "UNIX Network Programming". Prentice Hall, Inc., 1990.

[Stev94] W. R. Stevens. "TCP/IP Illustrated, Volume 1". Addison-Wesley Publishing Co., 1994.

[Whet95] B. Whetten, T. Montgomery, M. Kalfane. "A Fast Track Architecture for High Performance, Fault-Tolerant Distributed Group Communication". Work in progress.

[Wrig95] G. R. Wright and W. R. Stevens. "TCP/IP Illustrated, Volume 2". Addison-Wesley Publishing Co., 1995.